1. .OCT08 pwalk

To solve the problem, we find the lengths of the paths for each pair of nodes. We do this recursively by doing a depth-first search (DFS) or iteratively with a queue by doing a breadth-first search (BFS), starting from each node. (For more information, see either the training pages or the following links: [DFS](http://en.wikipedia.org/wiki/Depth-first_search) [BFS](http://en.wikipedia.org/wiki/Breadth-first_search).) Note that these searches will always find the paths in linear time for each node because the graph is a tree.

If implemented efficiently, this should take O(N2) time. Afterward, we output the answers to each of the queries one by one for an overall O(N2 + Q) solution. It is also possible to do a single DFS or BFS for each query, resulting in an overall O(NQ) solution that is also fast enough.

The following is a solution that uses BFS:

#include <cstdio>

#include <vector>

using namespace std;

FILE \*fout = fopen ("pwalk.out", "w");

FILE \*fin = fopen ("pwalk.in", "r");

struct edge

{

int past, dist;

edge (int p, int d)

{

past = p, dist = d;

}

};

const int MAXN = 1005;

int N, Q, A, B, L;

vector <edge> adj [MAXN];

int dist [MAXN][MAXN];

int q [MAXN];

int qhead, qtail;

// find the distances to all nodes from num

inline void findpaths (int num)

{

qhead = qtail = 0;

// initialize queue

q [qtail++] = num;

dist [num][num] = 0;

while (qhead < qtail)

{

int top = q [qhead++];

// go through each neighbor

for (int i = 0; i < (int) adj [top].size (); i++)

if (dist [num][top] + adj [top][i].dist < dist [num][adj [top][i].past])

{

// update distance and add node to queue

dist [num][adj [top][i].past] = dist [num][top] + adj [top][i].dist;

q [qtail++] = adj [top][i].past;

}

}

}

int main ()

{

// initialize to 'infinity'

memset (dist, 63, sizeof (dist));

// read input

fscanf (fin, "%d %d", &N, &Q);

for (int i = 1; i < N; i++)

{

fscanf (fin, "%d %d %d", &A, &B, &L);

// use 0-based indexing

A--, B--;

// add edges

adj [A].push\_back (edge (B, L));

adj [B].push\_back (edge (A, L));

}

// solve all paths

for (int i = 0; i < N; i++)

findpaths (i);

// output

while (Q--)

{

fscanf (fin, "%d %d", &A, &B);

A--, B--;

fprintf (fout, "%d\n", dist [A][B]);

}

return 0;

}

The following is a solution using DFS:

#include <cstdio>

#include <vector>

using namespace std;

FILE \*fout = fopen ("pwalk.out", "w");

FILE \*fin = fopen ("pwalk.in", "r");

struct edge

{

int past, dist;

edge (int p, int d)

{

past = p, dist = d;

}

};

const int MAXN = 1005;

int N, Q, A, B, L;

vector <edge> adj [MAXN];

int dist [MAXN][MAXN];

// find the lengths of the paths from a certain starting point

void search (int start, int num, int tot)

{

// already found the path, exit

if (tot >= dist [start][num])

return;

// set length of path

dist [start][num] = tot;

// go through all neighbors

for (int i = 0; i < (int) adj [num].size (); i++)

search (start, adj [num][i].past, tot + adj [num][i].dist);

}

int main ()

{

// initialize to 'infinity'

memset (dist, 63, sizeof (dist));

// read input

fscanf (fin, "%d %d", &N, &Q);

for (int i = 1; i < N; i++)

{

fscanf (fin, "%d %d %d", &A, &B, &L);

// use 0-based indexing

A--, B--;

// add edges

adj [A].push\_back (edge (B, L));

adj [B].push\_back (edge (A, L));

}

// solve all paths

for (int i = 0; i < N; i++)

search (i, i, 0);

// output

while (Q--)

{

fscanf (fin, "%d %d", &A, &B);

A--, B--;

fprintf (fout, "%d\n", dist [A][B]);

}

return 0;

}

B.FEB07 lilypad pond

It suffices to consider the grid as a graph and the possible jumps as edges. Then the problem becomes: Given a graph where some vertices are marked, find the least number of additional vertices to mark so there exist a path between s and t.

First notice that each connected component can be 'shrinked' into one vertex which combines all the edges of the vertices it contains as we could walk around the component free of cost.

The standard way to go here is to construct some kind of weight on the edges to change the problem into finding the shortest path and counting the number of shortest paths. It follows that the distance should be the least number of vertices that needs to be marked

Clearly, any edge from an unmarked vertex to an unmarked vertex has cost one as another vertex needs to be marked. Things gets tricky in situations where a marked vertex is traversed, namely the following situations:  
v1, v2 are marked, a and b are not. The following edges exist:  
a->v1, a->v2, v1->b, v2->b  
Then any kind of shortest path which involves v1 and v2 would end up calculating the same set of unmarked vertices (a and b) twice since there are two paths. Therefore, it's necessary to 'skip' marked vertices by having a path of length 1 directly from a to b.

It's quite clear that the least number of vertices to be marked equals the shortest distance from s to t minus one (the last edge to t should be free).

To get the answers, do a BFS from s, then go through each vertex in order and count the number of paths by summing the paths that lead to of all its neighbors which are distance d-1 away (assuming this vetex has distance d from s).

Here is 1000 point scorer Yang Yi's solution, which might or might not match the description above:

#include <stdio.h>

#include <queue>

#include <fstream>

#define MAXN 30

using namespace std;

FILE \*in;

int fx[8] = {-2,-1,+1,+2,+2,+1,-1,-2};

int fy[8] = {-1,-2,-2,-1,+1,+2,+2,+1};

int m,n,sx,sy,ex,ey;

int data[MAXN][MAXN];

int rea[MAXN][MAXN][MAXN][MAXN];

int dist[MAXN][MAXN];

long long counter[MAXN][MAXN];

queue<pair<int, int>> Q;

void init () {

int i, j;

in = fopen("lilypad.in","r");

fscanf(in,"%d%d",&m,&n);

for (i = 0; i < m; i ++)

for (j = 0; j < n; j ++) {

fscanf(in,"%d",data[i] + j);

if (data[i][j] == 3) {

data[i][j] = 0;

sx = i;

sy = j;

}

if (data[i][j] == 4) {

data[i][j] = 0;

ex = i;

ey = j;

}

}

fclose(in);

}

void solve () {

int i, j, x, y;

memset(rea, 0, sizeof(rea));

for (i = 0; i < m; i ++)

for (j = 0; j < n; j ++)

if (data[i][j] == 0) {

// printf("i j %d %d\n",i,j);

Q.push(make\_pair(i,j));

rea[i][j][i][j] = 1;

while (!Q.empty()) {

x=Q.front().first;

y=Q.front().second;

Q.pop();

for (int k = 0; k < 8; k ++)

if (x + fx[k] >= 0 && x + fx[k] < m && y + fy[k] >= 0 && y + fy[k] < n && data[x+fx[k]][y+fy[k]] == 1 && rea[i][j][x+fx[k]][y+fy[k]] == 0) {

Q.push(make\_pair(x + fx[k], y + fy[k]));

rea[i][j][x+fx[k]][y+fy[k]] = 1;

}

}

for (x = 0; x < m; x ++)

for (y = 0; y < n; y ++)

if (rea[i][j][x][y] == 1) {

// printf("Lily Reachable %d %d\n",x,y);

for (int k = 0; k < 8; k ++)

if (x + fx[k] >= 0 && x + fx[k] < m && y + fy[k] >= 0 && y + fy[k] < n && data[x+fx[k]][y+fy[k]] == 0 && rea[i][j][x+fx[k]][y+fy[k]] == 0)

rea[i][j][x+fx[k]][y+fy[k]] = 2;

}

}

memset(dist, -1, sizeof(dist));

dist[sx][sy] = 0;

counter[sx][sy] = 1;

Q.push(make\_pair(sx,sy));

while (!Q.empty()) {

x=Q.front().first;

y=Q.front().second;

Q.pop();

// printf("%d %d : %d %I64d\n",x,y,dist[x][y],counter[x][y]);

for (i = 0; i < m; i ++)

for (j = 0; j < n; j ++)

if (rea[x][y][i][j] == 2) {

// printf("(%d %d) -> (%d %d)\n",x,y,i,j);

if (dist[i][j] == -1) {

dist[i][j] = dist[x][y] + 1;

counter[i][j] = counter[x][y];

Q.push(make\_pair(i,j));

}

else

if (dist[i][j] == dist[x][y] + 1)

counter[i][j] += counter[x][y];

}

}

// while (1);

}

void output () {

ofstream fout("lilypad.out");

if (dist[ex][ey] == -1)

fout<<-1<<endl;

else

fout<<dist[ex][ey] - 1<<endl<<counter[ex][ey]<<endl;

}

int main () {

init();

solve();

output();

return 0;

}

C.FEB08 hotel

This problem is intended as an exercise in the mechanics of range tree. The information we need to track are fairly complicated, so it's best to break it into several operations on a 0,1 array:

\* Find the first group of k consecutive 1s  
\* Set an entire range to 0 (or 1).

The first operation can be further reduced to finding the longest segment of consecutive 1s in a range as we could search down the ranges. To track this, we need the following information in order to allow this information to propagate up the tree:

* maximum segment of 1s within the range
* maximum segment of 1s starting from the left point of the range
* maximum segment of 1s starting from the right point of a range

The 2nd and 3rd are fairly easy to propagate up the tree. The 1st is slightly trickier since there are 2 possibilities: range completely contained in left/right subrange, range crosses midpoint. In the 2nd case, the answer must be the maximum segment of 1s at the right end of the left sub-range and the maximum segment of 1s at the left end of the right sub-range merged together.

We also need to support 'fills'. To do this, we put a flag on each node to indicate whether it has been filled. Every time we update downwards from the root and we encounter these flags, we set the maximum values in that range accordingly and push the range onto its children as if a range if filled, so is any subrange of it.

All the above operations runs in time proportional to the height of the tree. So if we use an array-based binary indexed tree, we get O(logN) per query for a total runtime of O(MlogN).

Here is a working version of David Benjamin's code. The fields in the node structure are precisely the information listed above:

#include <assert.h>

#include <stdio.h>

#include <algorithm>

int realn;

int n;

enum { EMPTY, FULL, MIXED };

struct node {

char state;

int lmax;

int rmax;

int cmax;

int start;

unsigned size;

};

const int NMAX = 2\*160000+1;

const int ROOT = 1;

node tree[NMAX];

inline int left(int v) { return 2\*v; }

inline int right(int v) { return 2\*v+1; }

inline int parent(int v) { return v/2; }

inline unsigned size(int v) { return tree[v].size; }

inline int start(int v) { return tree[v].start; }

inline int end(int v) { return start(v) + size(v); }

void propogate(int v) {

if (left(v) >= 2\*n) return;

if (tree[v].state == EMPTY) {

unsigned s = size(left(v));

tree[left(v)].state = EMPTY;

tree[left(v)].lmax = s;

tree[left(v)].rmax = s;

tree[left(v)].cmax = s;

tree[right(v)].state = EMPTY;

tree[right(v)].lmax = s;

tree[right(v)].rmax = s;

tree[right(v)].cmax = s;

} else if (tree[v].state == FULL) {

tree[left(v)].state = FULL;

tree[left(v)].lmax = 0;

tree[left(v)].rmax = 0;

tree[left(v)].cmax = 0;

tree[right(v)].state = FULL;

tree[right(v)].lmax = 0;

tree[right(v)].rmax = 0;

tree[right(v)].cmax = 0;

}

}

void inherit(int v) { // assert - left & right are up-to-date

if (left(v) >= 2\*n) return;

if (tree[left(v)].state == EMPTY

&& tree[right(v)].state == EMPTY) {

tree[v].state = EMPTY;

tree[v].lmax = tree[v].rmax = tree[v].cmax = size(v);

return;

}

if (tree[left(v)].state == FULL

&& tree[right(v)].state == FULL) {

tree[v].state = FULL;

tree[v].lmax = tree[v].rmax = tree[v].cmax = 0;

return;

}

tree[v].state = MIXED;

tree[v].lmax = tree[left(v)].lmax;

if (tree[left(v)].state == EMPTY)

tree[v].lmax = tree[right(v)].lmax + size(left(v));

tree[v].rmax = tree[right(v)].rmax;

if (tree[right(v)].state == EMPTY)

tree[v].rmax = tree[left(v)].rmax + size(right(v));

tree[v].cmax = std::max(std::max(

tree[left(v)].cmax, tree[right(v)].cmax),

tree[left(v)].rmax + tree[right(v)].lmax);

}

int query(int d, int v) {

if (v >= 2\*n) return -1;

if (tree[v].state == EMPTY && tree[v].size <= d) return tree[v].start;

propogate(v);

if (tree[left(v)].cmax >= d) return query(d, left(v));

if (tree[left(v)].rmax + tree[right(v)].lmax >= d) return

end(left(v))-tree[left(v)].rmax;

if (tree[right(v)].cmax >= d) return query(d, right(v));

return -1;

}

void fill(int a, int b, int v) {

//printf("f %d %d\n", a,b);

int s = start(v);

int e = end(v)-1;

if (v >= 2\*n || b < a || s > b || e < a) return;

propogate(v);

if (s >= a && e <= b) {

tree[v].state = FULL;

tree[v].lmax = tree[v].rmax = tree[v].cmax = 0;

return;

}

fill(a,b, left(v));

fill(a,b, right(v));

inherit(v);

}

void empty(int a, int b, int v) {

//printf("e %d %d\n", a,b);

int s = start(v);

int e = end(v)-1;

if (v >= 2\*n || b < a || s > b || e < a) return;

propogate(v);

if (s >= a && e <= b) {

tree[v].state = EMPTY;

tree[v].lmax = tree[v].rmax = tree[v].cmax = size(v);

return;

}

empty(a,b, left(v));

empty(a,b, right(v));

inherit(v);

}

int main() {

FILE \* fin = fopen("hotel.in", "r");

FILE \* fout = fopen("hotel.out", "w");

assert(fin != NULL); assert(fout != NULL);

int m;

fscanf(fin, "%d %d\n", &realn, &m);

n = 1; while (n < realn) n \*= 2;

for (int i=n;i<2\*n;i++) {

tree[i].size = 1;

tree[i].start = i-n;

}

for (int i=n-1;i>0;i--) {

tree[i].size = 2\*tree[left(i)].size;

tree[i].start = tree[left(i)].start;

}

for (int i=1;i<2\*n;i++) {

tree[i].state = EMPTY;

tree[i].lmax = size(i);

tree[i].rmax = size(i);

tree[i].cmax = size(i);

}

fill(realn, n-1, ROOT);

for (int i=0;i<m;i++) {

int type;

fscanf(fin, "%d", &type);

if (type == 1) {

int d;

fscanf(fin, "%d", &d);

int p = query(d, ROOT);

fprintf(fout, "%d\n", p+1);

if (p >= 0)

fill(p, p+d-1, ROOT);

} else {

int x,d;

fscanf(fin, "%d %d", &x, &d); x--;

empty(x, x+d-1, ROOT);

}

}

return 0;

}

D.NOV08 cheering up the cows

Given a choice of the edges and starting point, which form a rooted tree, the optimal path is an Euler tour from the root. That is, we visit nodes in order:

visit(node v):

go to v

for c in children of v:

visit(c)

go to v

A node with K children appears in a tree's Euler tour K+1 times. For all nodes except the root, K+1 is also the degree of the node (whereas a root with K children has degree K). Thus, a node of degree D is visited D times, with the root being visited once more. Also, each edge is traversed twice: once going down and once going up.

Our total visiting time, for a tree rooted at vertex R with edges of length L\_i and vertices of degree D\_j and cost C\_j is 2(L\_1 + ... + L\_N-1) + (D\_1\*C\_1 + ... + D\_N\*C\_N) + C\_R.

We can now simplify the problem by computing the cheapest tree and picking the smallest C\_R independently. We also note that counting a vertex j D\_j times is counting it once for every edge connected to it. Equivalently, this is attributing to each edge i, the costs of the vertices at either end, S\_i and E\_i.

If we relabel an edge from S to E with length L with a weight of 2\*L + C\_E + C\_S, the cost of the tree is the sum of the edges. This is exactly the minimum-cost spanning tree problem, which can be solved in O(P log P) time with Kruskal's or Prim's algorithm.

#include <assert.h>

#include <stdio.h>

#include <limits.h>

#include <algorithm>

const int MAXV = 10000+5;

const int MAXE = 100000+5;

int v,e;

int cost[MAXV];

struct edge {

int a,b;

int l;

bool operator<(const edge& e) const { return l < e.l; }

};

edge edges[MAXE];

int uf\_rank[MAXV];

int uf\_parent[MAXV];

/\* Initialize the world with v elements: object i in set i \*/

void uf\_init(int v) {

for (int i=0;i<v;i++)

uf\_parent[i] = i;

}

/\* Find the set containing v \*/

int uf\_find(int v) {

if (v == uf\_parent[v]) return v;

return (uf\_parent[v] = uf\_find(uf\_parent[v]));

}

/\* Merge the sets of a and b \*/

void uf\_union(int a, int b) {

a = uf\_find(a); b = uf\_find(b);

if (uf\_rank[a] > uf\_rank[b]) {

uf\_parent[b] = a;

} else {

uf\_parent[a] = b;

if (uf\_rank[a] == uf\_rank[b])

uf\_rank[b]++;

}

}

int main() {

FILE \* fin = fopen("cheer.in", "r");

FILE \* fout = fopen("cheer.out", "w");

assert(fin != NULL); assert(fout != NULL);

/\* Input \*/

int ans = INT\_MAX;

fscanf(fin, "%d %d", &v, &e);

for (int i=0;i<v;i++) {

fscanf(fin, "%d", cost+i);

// sleep in the cheapest pasture

ans = std::min(ans, cost[i]);

}

for (int i=0;i<e;i++) {

fscanf(fin, "%d %d %d", &edges[i].a, &edges[i].b, &edges[i].l);

edges[i].a--; edges[i].b--;

// adjust the edge costs

edges[i].l \*= 2;

edges[i].l += cost[edges[i].a];

edges[i].l += cost[edges[i].b];

}

// visit edges in order of cost

std::sort(edges, edges+e);

/\* Kruskal's Algorithm \*/

uf\_init(v);

for (int i=0;i<e;i++) {

// don't connect anything already connected

if (uf\_find(edges[i].a) == uf\_find(edges[i].b)) continue;

// add edge i

ans += edges[i].l;

uf\_union(edges[i].a, edges[i].b);

}

fprintf(fout, "%d\n", ans);

return 0;

}